Turbo Macro Pro/TMPx Syntax

TMPx, Assembler Overview

TMPx source code syntax is a superset of it's c64-based sibling, Turbo Macro Pro. Basic elements of the syntax are:

labels: Something like variable names, a label is used to refer to a specific memory address or retain a value that can be used in an expression. Labels can be assigned a value like a memory address or just a numeric constant explicitly or are set by the assembler as it resolves unassigned labels.

opcodes: These are the standard 6502 mnemonics, in the form of three letter abbreviations. TMPx observes 6502 mnemonics as well as so-called illegal mnemonics and opcodes specific to the DTV v2 (aka PTV) joystick and wheel hardware.

pseudo ops: Assembler directives that instruct TMPx to perform some operation instead of merely translating an opcode into machine code.

comments: A simple way to annotate source code; comments are completely ignored by the assembler.

Turbo Macro Pro, Assembler Details

Constant Values

Constant values can be expressed in either decimal, hexadecimal, binary, or as characters depending on how the value is written. The largest constant value TMPx can recognize is $ffff (65,535).

```
   $  denotes a hexadecimal value
   %  denotes a binary value (limited to single bytes)
 'a'  denotes a character value, which translates to a byte
```

Any value that is not preceded by $ or % or wrapped in quotes is treated as a decimal value.

```
-=> Examples

 $20        = hex   $20, dec   32
 $2000      = hex $2000, dec 8192
 15         = hex   $0f, dec   15
 %10001000  = hex   $88, dec  135
 '1'        = hex   $31, dec   49
```

ASCII/PETSCII

TMPx assembles from ASCII source code files. This presents some challenge when developing software for the C64 which uses PETSCII. TMPx resolves this by allowing the use of tokenized representations of PETSCII as necessary. The token notation chosen is referred to as 'bastext', which is the name of a program developed in the 90's to help bridge PETSCII to ASCII when translating BASIC programs and that used either short text identifers or three digit (zero/left padded) decimal values inside curly brackets. TMPx recognizes bastext tokens:

```
-=> Examples

 {pound}    = british pound sign, PETSCII code 92
 {white}    = color white, PETSCII code 5
 {127}      = graphical character, PETSCII code 127
```

The bastext notation is further expanded to include any three digit decimal or hexidecimal value enclosed in curly braces. Such tokens are translated directly into the corresponding PETSCII code:

```
-=> Examples

 {$20}      = space, PETSCII code 32
 {063}      = question mark, PETSCII code 63
```

A bastext token can be used as a single character constant value enclosed in single quotes or as or or more characters in a double quoted string. Bastext tokens are referred to as either 'named' (they use a short text identifier) or un-named (they use decimal or hexidecimal). A full list of the named bastext tokens recognized by TMPx is given at the end of this documentation.

Labels

Valid labels in TMPx can be composed of any combination of letters, numbers, and the underscore character (obtained by pressing CBM+@). Labels must begin with a letter or underscore. Label names can be used in expressions (see below) in the same manner as constant values.

To define a label explicitly, set the label to either a constant value, to another label, or to an expression which will be resolved into an value. Once set, a label's value can not change. Attempting to define the same label twice will result in an error.

```
-=> Examples

 bah = $1000   ;sets the label 'bah' to $1000.
 lab = bah     ;sets the label 'lab' to equal whatever address bah
               ; resolves to (in this case, it would also be $1000).
 tmp = bah+$8  ;sets the label 'tmp' to equal the address that the
               ; expression 'bah+$8" resolves to ($1008).
```

There is a special label, which is called the 'program counter' and is represented in source code as the asterisk (*). It is important to set the value of * at the top of your code. When TMPx assembles your source, * is used to tell the assembler where the machine code should be assembled to. Set the value of * just like a label:

```
 *   = $1000   ;sets the program counter to $1000.
```

Note that unlike normal labels, you can reset the value of * anywhere you want in your source code. Each time you redefine *, the assembler simply uses the new value and continues assembling machine code at the new address.

Expressions

Expressions in TMPx can be formed with the following operators, all of which are binary operators (i.e. they require a value on the right AND left side of the operator):

```
 +   for addition
 -   for subtraction
 *   for multiplication
 /   for division
 &   for bitwise and
 .   for bitwise or
 :   for bitwise eor
```

Values used with expressions can be constant values described above, or labels. Note that you can also use parenthesis and as you would expect they affect the order in which the expression is evaluated:

```
-=> Examples

 $20 + 4          = hex $24, dec  36
 15 - %00000011  = hex $0c, dec  12
 $ff & $f0        = hex $f0, dec 240
 (2*$10)+1        = hex $21, dec  33
 2*($10+1)        = hex $22, dec  34
```

Unlike standard expressions in C/C++, there is no operator precedence. For example, multiplication has no higher precedence than addition:

```
 2*$10+1          = hex $21, dec  33
 1+2*$10          = hex $20, dec  32
```

In the first example above, the multiplication is resolved first, followed by the addition. In the next example, the addition is processed first! As this is a simplified way of resolving expressions than is more commonly seen in programming languages, it is important to keep in mind. Careful use of parenthesis can help make the expected results explicit.

The special '*' label can also be used in expressions, and it will always resolve to whatever the current program counter is where the assembler is outputting machine code to:

```
 *= $1000     ;sets the program counter to $1000
 jmp *+$03    ;assembles as jmp $1003
 jmp *        ;also assembles as jmp $1003 because the program counter
                at this line will actually be $1003.
```

The value of an expressions can also be modified by the following characters, which must appear at the very front of the expression:

```
 <    denotes that the low byte of the following constant or
       expression should be taken
 >    denotes that the hi byte of the following constant or
       expression should be taken
 !    denotes an expression whose value may currently or later
       be determined as needing only 1 byte (i.e. 0-255) but
       which should be expanded into 2 bytes for the purposes
       of outputting machine code. See below for an example.
```

```

-=> Examples

 >$0314           = hex $03
 <$0314           = hex $14
 <$0400+(6*$28)  = hex $f0
```

Examples of the use of '!' are shown below.

```
 lda $02    assembles as lda $02
 lda !$02   assembles as lda $0002
```

Another use of the '!' is to prevent phase errors. A phase error occurs when the length of code as determined by pass one and pass two of the assembler do not match. One way this can occur is by labels referenced before they are defined:

```
        lda bah  ; here, bah is undefined and TMPx assumes it is a word
        rts
  bah = $02      ; bah is now defined as a byte value
```

Assembling this code block results in a phase error, because TMPx assumes undefined labels represent words during the first pass. If the label is then found defined as a byte value, or if it actually resolves to a byte value, then the second pass will be shorter than the first and a phase error occurs. To prevent the phase error change the code to look like:

```
        lda !bah
        rts
  bah = $02
```

The '!' tells TMPx that the expression following should be treated as a word no matter how the label was defined. Obviously the best choice would be to define labels before referencing them but in our own experience this is sometimes undesired so using '!' notation will help avoid the woes of phase errors.

Comments

You can add comments to your source code by using the ';' (semicolon). Any ';' found by turbo will cause the assembler to ignore whatever follows the ';' til the end of that line. So comments can follow an opcode, or a pseudo-op, or they can be on a line by themselves.

```
-=> Examples

  ; this is a comment
 lda #$01   ; this is a comment
 sta $d020  ; so is this!
```

Pseudo-ops: Data

The following pseudo-ops are the standard way to make data tables. Note that values are not necessarily constrained to constants - they can be complete expressions as shown above.

```
 .byte - takes a list of byte values which can be decimal, hex, binary
         or character constants and produces a table of bytes.
 .word - takes a list of word values which can be decimal or hex and
         produces a table of bytes where the words are arranged in
         low-byte hi-byte order.
 .rta  - takes a list of word values which can be decimal or hex and
         produces a table of bytes where the words are decremented
         by 1 and set in low-byte hi-byte order; this is useful for
         doing stack manipulation with e.g. return addresses.
```

```
-=> Examples

 .byte 25,"a",$cc   =   $19 $41 $cc
 .word $fce2        =   $e2 $fc
 .rta $fce2         =   $e1 $fc
```

A similar set of pseudo-ops can be used to make string tables. Note that bastext tokens can be used inside strings.

```
 .text   - takes a quoted string and converts it into a span of bytes.
           Characters are converted from ASCII to PETSCII, including any
           bastest token conversion.
 .null   - as .text, but the span of bytes is terminated with
            a null (0) byte.
 .shift  - as .text, but the last byte has its high-bit set to 1.
 .screen - as .text, but the characters are converted into screencodes.
```

```
-=> Examples

 .text "hi"         =   $48 $49
 .null "hi"         =   $48 $49 $00
 .shift "hi"        =   $48 $c9
 .screen "hi"       =   $08 $09
```

Another pseudo-op is provided for convenience when you want tp specify a large span of repetitive values. .repeat with a count followed by a sequence of one or more values or expressions will create as many copies of the value/expression as you've specified.

```
 .repeat  - produce 1 or more copies of a sequence of 1 or more values.
```

```
-=> Examples

label = $9876

 .repeat 8,$ff           =   $ff $$ff $ff $ff $ff $ff $ff $ff
 .repeat 3,label         =   $76 $98 $76 $98 $76 $98
 .repeat 2,"a","b","c"  =   $41 $42 $43 $41 $42 $43
```

Pseudo-ops: Conditional Assembly

These pseudo-ops allow TMPx to assemble chunks of code based on the evaluation of an expression - if it is equal or not equal to zero or if it is positive or negative. You can use the functionality to selectively assemble code based on the value of a label, for example, or even the current value of the program counter.

```
 .if     - tests value or expression for inequality with zero.
            (same as .ifne)
 .ifne   - tests value or expression for inequality with zero.
            (same as .if)
 .ifeq   - tests value or expression for equality with zero.
 .ifpl   - tests value or expression for positive (the condition passes
           if the evaluated expression is from $0 - $7fff and fails
           if the evaluated expression is from $8000-$ffff).
 .ifmi   - tests value or expression for negative (the condition passes
           if the evaluated expression is from $8000 - $ffff and fails
           if the evaluated expression is from $0-$7fff).
 .endif  - marks the end of the code block that is assembled when the
            conditional test is met.
```

```
-=> Examples

cycle = 65
     .ifne cycle-65 ;if cycle != 65
     nop            ;produce one nop
     .endif
```

In addition to conditions based on the value of an expression, two other pseudo-ops test based simply on whether a label had been defined or not, regardless of its value. Both of the following pseudo-ops are also closed by .endif.

```
 .ifdef    - tests whether a label has been defined, condition passes if it has.
 .ifndef   - tests whether a label has been defined, condition passes if it has not.
```

```
-=> Examples

ntsc = 1
     .ifdef label
     nop            ;produce one nop
     .endif
```

Note that .ifdef and .ifndef can be most useful for testing whether a label definition has been specified in the command line arguments to TMPx. A scenario where this may be applied would be activating debugging code in output based on passing TMPx a "-D DEBUG" command line argument and testing for that in the source with ".ifdef DEBUG"

Pseudo-ops: Blocks (Scope Control)

The block pseudo-op allows a very useful operation: marking a section of code so that it can contain its own local labels, which can be redefined outside of the block or within a different block. Using blocks when coding reusable subroutines can ease their integration into new code... when the blocked subroutine is inserted into another source, a coder needn't worry that the labels used inside the subroutine block will conflict with any labels already defined in the other code. Note that blocks can be nested inside other blocks, each one creating a local scope for labels defined inside.

```
 .block  - starts a code block
 .bend   - ends a code block
```

```

-=> Examples

tmp = $02           ;tmp is defined as $02
     lda tmp        ;assembles as  lda $02
sub                 ;our subroutine label name, 'sub'
     .block
tmp = $ff           ;tmp is defined as $ff but only applies in the block
     clc
     adc tmp        ;assembles as  adc $ff, not adc $02!
     rts
     .bend
     lda tmp        ;assembles as  lda $02 again...
```

Pseudo-ops: Variables

Variables are a special kind of label that can be redefined (even without worrying about localized blocks). This is the only way to change the value of a label in the same context without getting a 'double defined' error. These are most useful when combined with other pseudos like conditionals (see above) and goto (see below).

```
 .var - defines a variable identified by the label preceeding '.var',
        which is set to the value of the expression following the '.var'.
```

```

-=> Examples

va    .var $01   ;defines a var named va with value of 1
      lda #va    ;results in lda #$01
va    .var va+1  ;redefines va to it's current value+1
      lda #va    ;results in lda #$02
```

Pseudo-ops: Unconditional Goto

These pseudo-ops force the assembler to jump to a given label and continue assembling from there. This can be extremely useful for repeating a short code block 10, 100, or even more times. Only labels flagged with a .lbl psuedo-op can be referenced by a .goto!

```
 .lbl   - identifies a label that can be referenced by a .goto
 .goto  - causes assembler to goto the label referenced and continue
          assembling from that point on.
```

```

-=> Examples

cnt    .var $0100  ;cnt = $100
loop   .lbl        ;label 'loop' is prepared to be a .goto target
       nop
cnt    .var cnt-1  ;dec cnt
       .ifne cnt   ;if cnt != 0
       .goto loop  ;goto loop
       .endif      ;until all $100 nop's are assembled
```

Pseudo-ops: Macros

Macros are a means of inserting a larger span of code by using a short macro label followed by 0 to 8 arguments. This "macro call" is expanded into the code at assembly time, with substitutions made for the arguments given. A macro call is identified by a '#' followed by the macro label.

```
 .macro    - starts a macro definition that will implicitly treat the expanded
             code block as though it were enclosed by .block/.bend.
             around the macro
 .segment  - starts a macro definition but without any implied block
 .endm     - ends a macro definition
```

Macro argument substitutions are denoted in the macro code with either a '\' (ASCII backslash) to mark a numeric substitution or an '@' symbol to mark a string substitution.

```
-=> Examples

poke   .macro    ;start macro def
       lda #\2   ;the \2 means an argument substitution, in this case the
                 ; second macro argument will replace the \2
       sta \1    ;another substitution
       .endm     ;end macro def
```

In the source code, a macro call for the above macro definition would look like:

```
       #poke $d020,0
```

Upon assembly the macro call is expanded, and substitutions are made (in this case \1 is replaced by $d020 and \2 is replaced by 0, resulting in the following assembly being generated to replace the macro call:

```
       lda #$00
       sta $d020
```

Here is a more complex example showing recursion with macros:

```
table .macro     ;start macro def
cnt   .var \1+1  ;set var cnt to arg 1 plus 1
      #tab       ;a macro call within a macro!
      .endm      ;end macro def

tab   .segment   ;start macro def
cnt   .var cnt-1 ;decrement cnt
      .if cnt    ;if cnt != 0
      #tab       ;then recursively macro call itself
      .endif     ;end of the if
      .byte cnt  ;assemble a byte with current value of cnt
cnt   .var cnt+1 ;increment cnt
      .endm      ;end macro def

       #table 64 ;call the macro to create a byte table from 0 to 64
```

Here is an example of using a text parameter in a macro definition and call:

```
error  .macro      ;start macro def
       lda #<tx    ;lo-byte of tx
       ldy #>tx    ;hi-byte of tx
       jsr $ab1e   ;print it
       jmp end     ;skip text data
tx     .null "@1"  ;@1 is replaced by a textual argument
end    .endm       ;end macro def

       #error "doc too boring"
```

So you see, use \x for numerical arguments, and @x for text arguments, where x denotes the argument number.

Text arguments can be even used to modify an opcode! Example:

```
do     .segment
       .block
loop   .endm

while  .segment    ;start macro def
       b@1 loop    ;the @1 will be replaced by a text argument
       .bend
       .endm       ;end macro def

       ldx #5      ;start of code
       #do         ;call macro do
       ldy #2
       lda #"*"
       #do         ;call macro do
       jsr $ffd2
       dey
       #while "pl" ;call macro 'while' using parameter "pl"
       lda #" "    ;print space
       jsr $ffd2
       dex
       #while "ne" ; call macro 'while' with parameter "ne"
```

Pseudo-ops: Includes

The .include psuedo-op allows you to assemble code from a different source file. For example, you could use .include to assemble in a file with label definitions for all of the kernal jumptable, or you could write small reusable subroutines separately and include them in as you need them.

```
 .include - load and assemble the specified file
```

```
-=> Example

 .include "kernel.s"
```

The .binary psuedo-op will include the binary data in the given file directly into the assembled output at the location of the .binary call. You may optionally specify a number of data bytes to skip from the start of the included file by specifying that after the filename. This would be used to, for example, skip a two byte load address at the start of the file.

```
 .binary - load and output bytes from the specified file, optionally
           skipping bytes at the start of the file
```

```
-=> Example

 .include "3d.dat"
 .include "music.prg",2
```

Pseudo-ops: Code Offsets

The .offs pseudo op is used to alter the memory address to where code is being assembled. This sounds identical to the function of the program counter (*) but there is a distinction: * sets the "effective address" for assembled code, meaning, object code is generated assuming that it is located at the address in the program counter. But .offs changes the "actual address" to where object code is assembled. If you do not use calls to .offs in your code, then the actual address and effective address will remain identical. Used together, you can assemble code so that it will execute properly in a different part of memory than where the code itself is located in the assembled output.

```
 .offs - change the location that code is assembled to; code will be assembled forward
         in memory the the value of the expression following .offs
```

```
-=> Example

     (this code)   ->  (assembles as)
       * = $2000
start  bit base0       bit $2009
       bit base        bit $8000
       jmp *           jmp $2006
base0  * = $8000
base   .offs base0-*
       lda #>start     lda #$20
       jmp *           jmp $8002
```

This code is assembled from $2000 to $200d; by using an expression for the .offs we can generate code in a contiguous memory span, even though part of the code has been assembled to execute at a different memory address (in this case, $8000). In this example the .offs is effectively offsetting the assembled code backwards in memory. Naturally this can be extremely useful for coding routines that need to execute in zero page, or for coding routines to run in drive memory.

Pseudo-ops: Printer Control

The printer psuedo-ops allow some control of output during assembly. .proff and .pron allow you to skip printing sections of code, while .hidemac and .showmac let you control whether the expanded macro gets printed or just the macro call.

```
 .pron     - turn on printing
 .proff    - turn off printing
 .hidemac  - show unexpanded calls
 .showmac  - show expanded calls
```

Pseudo-ops: Miscellaneous

```
 .eor      - followed by some value, with which all code after the
              .eor is eor'd with.
 .end      - terminate assembling
 .bounce   - try ".bounce 2,256,0,255,1,0,0,100,100,0,0" for a good time.
```

Site & Content © 2005-2012, Style.
style64.org